T-FSM: A Task-Based System for Massively Parallel Frequent Subgraph Pattern Mining from a Big Graph

LYUHENG YUAN* and DA YAN*, The University of Alabama at Birmingham, USA
WENWEN QU, East China Normal University, China
SAUGAT ADHIKARI, The University of Alabama at Birmingham, USA
JALAL KHALIL, The University of Alabama at Birmingham, USA
CHENG LONG, Nanyang Technological University, Singapore
XIAOLING WANG, East China Normal University, China

Finding frequent subgraph patterns in a big graph is an important problem with many applications such as classifying chemical compounds and building indexes to speed up graph queries. Since this problem is NP-hard, some recent parallel systems have been developed to accelerate the mining. However, they often have a huge memory cost, very long running time, suboptimal load balancing, and possibly inaccurate results. In this paper, we propose an efficient system called T-FSM for parallel mining of frequent subgraph patterns in a big graph. T-FSM adopts a novel task-based execution engine design to ensure high concurrency, bounded memory consumption, and effective load balancing. It also supports a new anti-monotonic frequentness measure called Fraction-Score, which is more accurate than the widely used MNI measure. Our experiments show that T-FSM is orders of magnitude faster than SOTA systems for frequent subgraph pattern mining. Our system code has been released at https://github.com/lyuheng/T-FSM.

CCS Concepts: • **Theory of computation → Graph algorithms analysis**; **Parallel algorithms**; • **Mathematics of computing → Graph algorithms**.

Additional Key Words and Phrases: frequent subgraph, graph, parallel, task, Fraction-Score

ACM Reference Format:
Lyuheng Yuan, Da Yan, Wenwen Qu, Saugat Adhikari, Jalal Khalil, Cheng Long, and Xiaoling Wang. 2023. T-FSM: A Task-Based System for Massively Parallel Frequent Subgraph Pattern Mining from a Big Graph. *Proc. ACM Manag. Data* 1, 1, Article 74 (May 2023), 26 pages. https://doi.org/10.1145/3588928

1 INTRODUCTION

The Problem. Frequent subgraph pattern mining (FSM) finds all subgraph patterns that appear more frequently than a given threshold in a graph database. FSM is essential for knowledge discovery from graph data such as biological networks [38] and social networks. The extracted patterns can be used as features for classifying chemical compounds [8] and for building indexes to

*Both authors contributed equally to this research.

Authors' addresses: Lyuheng Yuan, lyuan@uab.edu; Da Yan, yanda@uab.edu, The University of Alabama at Birmingham, Birmingham, Alabama, USA; Wenwen Qu, East China Normal University, Shanghai, China, wenwenqu@stu.ecnu.edu.cn; Saugat Adhikari, The University of Alabama at Birmingham, Birmingham, Alabama, USA, saugat@uab.edu; Jalal Khalil, The University of Alabama at Birmingham, Birmingham, Alabama, USA, jalalk@uab.edu; Cheng Long, Nanyang Technological University, Singapore, c.long@ntu.edu.sg; Xiaoling Wang, East China Normal University, Shanghai, China, xlwang@cs.ecnu.edu.cn.

2836-6573/2023/5-ART74 $15.00
https://doi.org/10.1145/3588928

speed up graph queries [37]. Other applications include graph clustering [24], protein functionality prediction [5], privacy preservation [39] and image processing [6].

FSM has 2 problem settings: finding frequent patterns either (i) in a database comprising many graph transactions (e.g., a set of chemical compounds), or (ii) from a single big input graph (e.g., a social network or a PPI network). In the transactional setting, FSM enumerates all subgraph patterns that appear in $\geq \tau$ transactions, where τ is a user-defined support threshold. However, in a single-graph setting, subgraph frequency violates the anti-monotonicity principle required for effective pattern pruning in FSM algorithms. Consider graph G_1 in Figure 1: subgraph pattern S_1 contains only a vertex labeled A, so it matches to only **1** data vertex v_1 in G_1. Also, subgraph pattern S_2 consists of two vertices labeled A and B, respectively, connected by an edge; it is a super-pattern to S_1, but it has **3** isomorphisms in G_1: (v_1, v_2), (v_1, v_3) and (v_1, v_4).

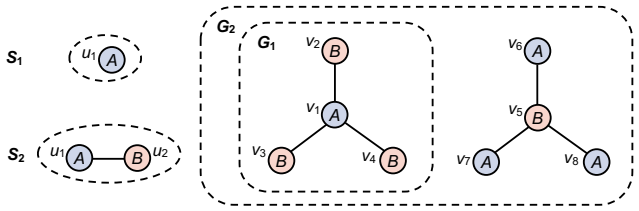

Fig. 1. Subgraph Patterns S_1, S_2 in Two Data Graphs G_1, G_2

Several anti-monotonic support measures have been proposed in the single-graph setting, among which only *minimum image* (MNI) [2] is computationally tractable (other measures are NP-complete [10]) and hence it becomes the de facto standard adopted by all prior works including ScaleMine [23], DistGraph [9], Arabesque [28], RStream [31], Fractal [11], Pangolin [4] and Peregrine [16]. Next, we illustrate the concept of MNI. We say that a data vertex $v \in G$ is a valid match to a pattern vertex $u \in S$, denoted by $v \rightsquigarrow u$, iff there exists an isomorphism of subgraph pattern S in G that contains v, where u is mapped to v. Then, ***given a subgraph pattern*** S ***containing vertices*** $u_1, u_2, \ldots, u_k$, ***the MNI of*** S ***measures the least number of distinct valid matches of all*** $u_i \in S$**.** For example, for pattern S_2 and graph G_1 in Figure 1, u_1 has one valid match v_1, while u_2 has three valid matches v_2, v_3 and v_4, so the MNI of S_2 in G_1 is $\min\{1, 3\} = 1$, which is more reasonable than the frequency support of value 3, since we can only find one pair of connected vertices with labels A and B in G_1.

However, we notice that MNI often significantly overestimates the true support. For example, consider pattern S_2 in G_2, where u_1 has four valid matches v_1, v_6, v_7 and v_8, and u_2 has four valid matches v_2, v_3, v_4 and v_5, so the MNI of S_2 in G_2 is $\min\{4, 4\} = 4$, but only two pairs of connected vertices with labels A and B can occur at any time in G_2. Figure 17 in Section 4 empirically shows how much MNI overestimates the true support. Surprisingly, this overestimation issue of MNI did not raise attention in prior works that unanimously used MNI for support. We overcome this issue by generalizing the Fraction-Score [3] measure from the context of co-location pattern mining to our single-graph setting, which can recover the ideal support value of 2 in the above example. Section 2.2 will explain the definition of Fraction-Score in our context and how it is different from Fraction-Score in co-location pattern mining [3].

This paper focuses on the single-graph setting. Solutions to transactional settings have been well studied including algorithms gSpan [36], Gaston [18] and the parallel PrefixFPM [35, 34] system.

Algorithm Framework. In the single-graph FSM setting, GraMi [10] is the state-of-the-art serial mining algorithm, which we illustrate using the example in Figure 2. Specifically, for each vertex u in pattern S, we define the domain of u, denoted by $D(u)$, to be the set of candidate data vertices in G that u can be mapped to. For example, $D(u_1) = \{v_1, v_4, v_7, v_8\}$ in Figure 2 since these vertices have the same label A. Here, GraMi would conduct 4 subgraph matching operations of S in G, one for each vertex v in $D(u_1)$ assuming that u_1 has been mapped to v. Each subgraph matching operation returns by confirming $v \rightsquigarrow u_1$ as soon as a matched subgraph in G is found, without enumerating

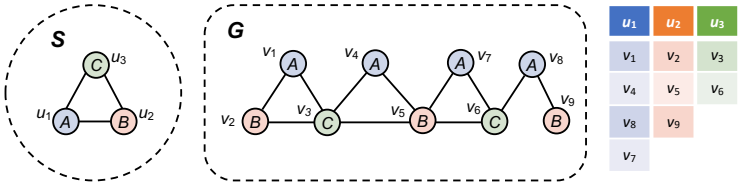

Fig. 2. Domain Illustration

all matched subgraph instances as in a regular subgraph matching operation. In contrast, if no matched subgraph is found, candidate v is not a valid match to u_1 (denoted by $v \not\rightsquigarrow u_1$). Referring to Figure 2 again, we can see that $v_1, v_4, v_7 \rightsquigarrow u_1$ but $v_8 \not\rightsquigarrow u_1$, so u_1 has 3 valid matches and we say that its valid domain is $D^*(u_1) = \{v_1, v_4, v_7\}$. Similarly, we obtain $D^*(u_2) = \{v_2, v_5\}$ and $D^*(u_3) = \{v_3, v_6\}$, so the MNI of S is $\min\{3, 2, 2\} = 2$.

Given an MNI support threshold τ, GraMi allows early termination when checking a domain $D(u)$ as soon as τ valid matches have been found. For example, in Figure 2, if $\tau = 2$ and we already find $v_1 \rightsquigarrow u_1$ and $v_4 \rightsquigarrow u_1$, we do not need to continue checking v_7 and v_8 in $D(u_1)$ but can move on to examine $D(u_2)$ and $D(u_3)$, since u_1 will not cause MNI to be less than τ (i.e., cause S to be infrequent). This early termination is the key reason why GraMi is efficient, since $|D^*(u_1)|$ can be much larger than τ in a real dataset.

A lot of pruning opportunities are possible in this algorithm framework. For example, in Figure 2 when examining $D(u_1)$, if the subgraph matching from v_1 finds a match to S, i.e., $\{v_1, v_2, v_3\}$, we can also add v_2 (resp. v_3) to $D^*(u_2)$ (resp. $D^*(u_3)$), so that later when examining $D(u_2)$ (resp. $D(u_3)$), we can skip subgraph matching from v_2 (resp. v_3) and conclude that $v_2 \rightsquigarrow u_2$ (resp. $v_3 \rightsquigarrow u_3$). As another example, let τ be 4, and assume we have checked v_1, v_4 and v_8 in $D(u_1)$ in Figure 2, then we have two valid matches v_1 and v_4 and can conclude that $|D^*(u_1)| < \tau$ without conducting subgraph matching from the last vertex v_7 in $D(u_1)$ (as even if $v_7 \rightsquigarrow u_1$, the support to u_1 is at most $1 + 2 < \tau$). Also, once $|D^*(u_1)| < \tau$, we know that S is infrequent without checking $D(u_2)$ and $D(u_3)$.

Parallelization. Since FSM is computationally expensive (NP-hard), some recent parallel systems have been developed to accelerate FSM in the single-graph setting. However, most of them (e.g., DistGraph [27] and Fractal [22]) grow and materialize patterns and all their matched instances in G, so they fail to utilize the above pruning opportunities and often incur a huge memory cost, limiting their scalability. Even though ScaleMine [1] conducts pruning, it uses some approximate approaches that treat patterns as frequent (resp. infrequent) as long as they are frequent (resp. infrequent) with a high probability, but we find that the results are often not accurate and could be inconsistent even in different runs. Also, previous works mainly focus on mining small-sized patterns, since their subgraph matching procedure is inefficient. We observed prohibitively long running time or out-of-memory error when they mine patterns with ≥ 6 vertices. However, some advanced subgraph matching algorithms have emerged recently [25, 26, 17] that are much more efficient. Therefore, it is important to integrate the latest subgraph matching techniques in FSM.

In this paper, we propose an efficient task-based parallel system, called T-FSM, for FSM in a single big graph. T-FSM enumerates subgraph patterns using the redundancy-free rightmost path extension technique from gSpan [36], and for each pattern S with vertices $u_1, u_2, \ldots, u_k$, the basic unit of execution is a task that conducts subgraph matching from some data vertex $v \in D(u_i)$, $i = 1, 2, \cdots, k$, using the latest subgraph matching algorithm [25]. Note that each pattern has many tasks, and these fine-grained tasks allow massive parallelization and effective load balancing. Our

underlying execution engine is designed to ensure all computing threads are kept busy computing the available tasks without the straggler problem.

There are a few challenges in implementing this task-based scheme efficiently. For example, (1) *how can we ensure low memory consumption?* T-FSM makes sure that at most n_{active}^{max} subgraph patterns are under computation at any time, where n_{active}^{max} is a user-specified parameter to limit memory usage. T-FSM only conducts subgraph matching to examine the frequentness of a new subgraph pattern when memory space permits (e.g., an existing pattern finishes processing). Moreover, each active pattern only maintains a limited number of subgraph-matching tasks from its domain table that are just sufficient to keep all the computing threads of T-FSM busy. This is in contrast to systems like DistGraph [27] that keep and process numerous patterns at the same time.

As another example, since each pattern S has many active tasks in computation, (2) *if one task determines that S is frequent (resp. infrequent) using the pruning rules described previously, how can we terminate the other tasks in time?* T-FSM maintains the latest status of each pattern currently under computation, so that its tasks can check the pattern status to avoid wasted computation.

Also, we find that subgraph matching from some data vertices $v \in D(u)$ can be much more expensive than others, causing the straggler problem in parallel execution. (3) *How can we eliminate straggler tasks?* For each pattern S, T-FSM initially puts those subgraph-matching tasks that run beyond a certain time threshold τ_{time} to a temporary buffer, in hope that the results from the other tasks can already determine that S is frequent; if not, those timeout tasks will be fetched back from the buffer to continue their computation. Moreover, a timeout task is allowed to time out again, in which case it will be decomposed into some smaller tasks to avoid becoming a straggler.

The main contributions of this work are summarized as follows:

- T-FSM uses a novel system design that ensures full CPU core utilization to compute fine-grained subgraph-matching tasks with bounded memory consumption, while utilizing advanced pruning techniques and effective load balancing enabled by a task-timeout mechanism.
- T-FSM integrates the state-of-the-art subgraph matching algorithm, allowing us to mine much larger patterns than existing systems, and to mine patterns in much less time.
- We indicate the support overestimation issue of the current de facto standard support measure, MNI, and generalize the concept of Fraction-Score in co-location pattern mining as a more accurate measure which is integrated in T-FSM.
- We conduct extensive experiments on 10 real graphs with diverse characteristics, which show that T-FSM significantly outperforms existing systems in terms of running time.

The rest of this paper is organized as follows. Section 2 first formally defines our FSM problem, describes our new Fraction-Score measure, and reviews the related work. Section 3 then describes the system design of T-FSM. Finally, Section 4 reports our experiments and Section 5 concludes this paper and discusses the future work.

2 PRELIMINARIES

This section first formally define our single-graph FSM problem and the useful notations in Section 2.1. Section 2.2 then introduces our new and more accurate support measure Fraction-Score. Finally, Section 2.3 reviews related work on subgraph matching and FSM.

2.1 Problem Definition

Without loss of generality, we consider an undirected graph $G = (V^G, E^G, L^G)$ with a vertex set V^G, an edge set $E^G \subseteq V^G \times V^G$, a label set L^G for vertices and edges. We only consider simple graphs without self-loops and multiple edges. Our algorithms can be easily generalized to a directed graph, as we shall discuss in Section 5.

Given a query graph S, **subgraph matching** finds all isomorphisms of S in data graph G, i.e., to find all mappings $\psi : V^S \rightarrow V^G$, such that (1) for each $u \in V^S$, we have $L^S(u) = L^G(\psi(u))$, and (2) for each $e = (u_i, u_j) \in E^S$, there exists $(\psi(u_i), \psi(u_j)) \in E^G$ and $L^S(e) = L^G(\psi(e))$. As an illustration, consider query graph S_2 and data graph G_1 in Figure 1, where A and B are vertex labels. Then, S_2 has 3 isomorphisms in G_1, namely (v_1, v_2), (v_1, v_3) and (v_1, v_4).

Given a **support threshold** τ, FSM in G finds all subgraph patterns S with support $\geq \tau$, where support is an anti-monotonic measure such as MNI [10] or Fraction-Score (see Section 2.2). Recall that we say that a data vertex $v \in G$ is a valid match to a pattern vertex $u \in S$, denoted by $v \rightsquigarrow u$, iff there exists an isomorphism of subgraph pattern S in G that contains v, where u is mapped to v. Also recall from Figure 2 that each pattern S is associated with a domain table, which maintains a column $D(u)$ of candidate data vertices to match to u for each $u \in S$. **MNI** [10] is a popular anti-monotonic support measure for single-graph FSM, which measures the least number of valid matches of every vertex $u \in S$, i.e., $mni(S) = \min_{u \in S} |D^*(u)|$. A pattern S is said to be frequent iff $mni(S) \geq \tau$.

2.2 Fraction-Score

We now define a more accurate support measure, Fraction-Score, in our single-graph FSM setting. Fraction-Score was originally proposed for mining co-location patterns from a single big spatial database, and has been proved to be anti-monotonic [3], and more accurate than other measures. We will explain our difference in the end; the anti-monotonicity proof is similar and thus omitted.

Let us denote the set of neighbors of a vertex $v \in G$ (resp. $u \in S$) by $N^G(v)$ (resp. $N^S(u)$), and denote those neighbors with label ℓ by $N^G_\ell(v)$ (resp. $N^S_\ell(u)$). Recall from Section 1 that pattern S_2 in Figure 1 has $mni(S_2) = 4$ which overestimates the ground-truth support 2. Our new Fraction-Score measure to be described below ideally addresses this problem and recovers the true support.

The problem with MNI is that (v_1, v_2), (v_1, v_3) and (v_1, v_4) all contribute 1 to the support of S_2 in Figure 1, while there is only one vertex v_1 with label A shared by 3 vertices with label B. Ideally, each of them should contribute only 1/3 to the support, since the contribution of v_1 is split by the 3 vertices v_2, v_3 and v_4. Let us define the fraction of $v' \in N^G(v)$ that a vertex $v \in G$ receives by

$$\Delta_{v'}(v) = \frac{1}{\left|N^G_{L^G(v)}(v')\right|}. \tag{1}$$

For example, in Figure 1, $\Delta_{v_1}(v_2) = 1/3$ because $L^G(v_2) = B$ and $N^G_B(v_1) = \{v_2, v_3, v_4\}$. Next, we define the total fractional contribution to v from all its neighbors with label ℓ by

$$\Delta_\ell(v) = \min\left\{\sum_{v' \in N^G_\ell(v)} \Delta_{v'}(v), \quad 1\right\}, \tag{2}$$

where we bound the total contribution by 1 since v should not contribute more than 1 to the support. For example, in Figure 1,

$$\Delta_B(v_1) = \min\{1 + 1 + 1, 1\} = 1 \tag{3}$$

since $\Delta_{v_2}(v_1) = \Delta_{v_3}(v_1) = \Delta_{v_4}(v_1) = 1$; while

$$\Delta_B(v_6) = \Delta_B(v_7) = \Delta_B(v_8) = \frac{1}{3} \tag{4}$$

since v_5 is the only neighbor of v_6, v_7 and v_8 with label B, and $\Delta_{v_5}(v_6) = \Delta_{v_5}(v_7) = \Delta_{v_5}(v_8) = \frac{1}{3}$.

Given a graph G, we preprocess it by computing $\Delta_{v'}(v)$ for every $v \in G$ and every neighbor $v' \in N^G(v)$, and then using them to compute $\Delta_\ell(v)$ for every $v \in G$ and every label ℓ (if $N^G_\ell(v) \neq \emptyset$).

Later during FSM, given a pattern S with vertices $\{u_1, u_2, \ldots, u_k\}$, we next define the fractional contribution to each data vertex $v \in D^*(u_i)$, $i = 1, 2, \cdots, k$ (i.e., v is a valid match to u_i). For such a vertex v, we consider the contributions to be coming from all its neighbors v' that match some vertex $u_j \in N^S(u_i)$ (already captured by $\Delta_{L^S(u_j)}(v)$ defined in Eq (2)). Specifically, the contribution to $v \in D^*(u_i)$ given a pattern S is defined to be the minimum fractional contribution to v among all such neighbors v' as follows:

$$\Delta^S(v) = \min_{u_j \in N^S(u_i)} \Delta_{L^S(u_j)}(v), \tag{5}$$

To illustrate, for pattern S_2 in Figure 1 and vertex $v_1 \in D^*(u_1)$, we have $N^S(u_1) = \{u_2\}$ and $L^S(u_2) = B$, so

$$\Delta^S(v_1) = \Delta_B(v_1) = 1 \tag{6}$$

(recall Eq (3)); and for pattern S_2 and vertex $v_2 \in D^*(u_2)$, we have $N^S(u_2) = \{u_1\}$ and $L^S(u_1) = A$, so

$$\Delta^S(v_2) = \Delta_A(v_2) = \frac{1}{3} \tag{7}$$

(recall Eq (4) and the symmetry of the two components of G_2).

Now that we have defined the contribution from each $v \in D^*(u_i)$ to pattern S (i.e., Eq (5)), we define the total contribution of $u_i \in S$ to pattern S as the sum of contributions from all $v \in D^*(u_i)$:

$$\sigma_{u_i}(S) = \sum_{v \in D^*(u_i)} \Delta^S(v). \tag{8}$$

For example, for G_2 in Figure 1, $\sigma_{u_1}(S_2) = \Delta^S(v_1) + \Delta^S(v_6) + \Delta^S(v_7) + \Delta^S(v_8) = 1 + \frac{1}{3} + \frac{1}{3} + \frac{1}{3} = 2$ since $\Delta^S(v_1) = 1$ (recall Eq (6)) and $\Delta^S(v_6) = \frac{1}{3}$ (recall Eq (7) and symmetry between v_2 and v_6).

Finally, as in MNI, the Fraction-Score of pattern S takes minimum among the total contributions of all pattern vertices $u_i \in S$:

$$FS(S) = \min_{u_i \in S} \sigma_{u_i}(S). \tag{9}$$

For G_2 in Figure 1, $FS(S_2) = \min\{\sigma_{u_1}(S_2), \sigma_{u_2}(S_2)\} = \min\{2, 2\} = 2$. Note that $FS(S_2)$ matches the exact ground truth that we expect!

Contributions and Differences from [3]. We remark that our Fraction-Score definition is a non-trivial generalization of that in co-location pattern mining [3] (CPM) for the following reasons.

Firstly, in single-graph FSM, MNI is the de factor standard support measure adopted by all prior works including ScaleMine [23], DistGraph [9], Arabesque [28], RStream [31], Fractal [11], Pangolin [4] and Peregrine [16], despite its support overestimation issue.

Secondly, in CPM, we are given k categories of POIs (points-of-interest) with different labels ℓ_1, ℓ_2, $\ldots$, ℓ_k (e.g., restaurants, hotels, banks, outlets), and the goal is to find subsets of labels such that their POIs co-occur (i.e., with pairwise distance $\leq d$) frequently in a POI dataset, where d is a user-defined distance parameter. There does not exist an explicit graph structure, so our definition needs to be properly redesigned for the graph context. For example, our Eq (5) defines the contribution of pattern S to v (where $v \rightsquigarrow u_i$) based on u_i's neighbors in subgraph pattern S, while the CPM counterpart (Eq (3) of [3]) is defined simply based on all other labels that are not the label of POI v in the POI label set S (i.e., pattern S).

Finally, even if we reformulate CPM as an FSM problem, CPM is still just a special case. Specifically, if we create an edge between any two POIs within distance d, then CPM basically mines the resulting graph G (called POI proximity graph) for frequent clique patterns where vertices (or, objects) have different labels. For example, our FSM algorithm can find a pattern S like the one in Figure 3 from the POI proximity graph G, where (i) on one day a tourist wants to visit a theme park and then to have dinner at a restaurant, so wants to find a hotel that co-occurs with a theme park and a

restaurant; and (ii) on another day the tourist wants to go shopping at an outlet where he/she can find an ATM nearby to withdraw money and find a restaurant to have lunch, so he/she wants these POIs and the hotel to co-occur; there is, however, no need (1) for the outlet and the theme park to be within distance d, and no need (2) for the two restaurants to be the same restaurant; both relaxations are not possible for a co-location pattern.

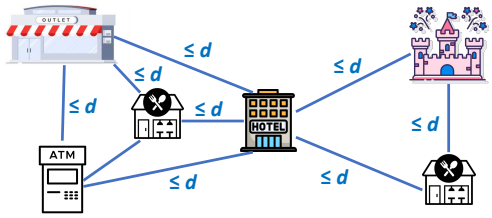

Fig. 3. A POI Pattern

The anti-monotonicity proof of our extended Fraction-Score measure is omitted since it is similar to that by [3] in its Lemma 1.

2.3 Related Work

In this subsection, we first review the state-of-the-art (SOTA) parallel FSM systems and explain their weaknesses. Table 1 summarizes the features of these systems and compares them with our T-FSM system. We then present the SOTA subgraph pattern enumeration algorithm gSpan [36] and the SOTA subgraph matching algorithm by [25], which are two primitives used by T-FSM.

Table 1. Feature Comparison of FSM systems

	0. In-memory task pool for timely computation	1. Bounded memory consumption	2. Fine-grained Load balancing	3. Result Exactness	4. SOTA subgraph matching algorithm	5. Task independence	6. GraMi pruning rules
T-FSM	✓	✓	✓	✓	✓	✓	✓
ScaleMine	✓	✓	x	x	x	✓	x
DistGraph	x	x	x	✓	N/A	x	x
Fractal	✓	✓	x	✓	N/A	x	x
Arabesque	✓	x	x	✓	N/A	x	x
Pangolin	✓	x	x	✓	N/A	x	x
Peregrine	✓	✓	x	✓	x	✓	x

ScaleMine. ScaleMine [1] solves FSM in two phases. The first phase is approximate, which (1) quickly identifies subgraph patterns that are frequent or infrequent with high probability, and for the remaining patterns, it (2) collects statistics to estimate the mining loads for the purpose of load balancing in the second phase. To estimate if a pattern S is frequent or not, for each $u_i \in S$, ScaleMine samples some candidates from $D(u_i)$ to perform subgraph matching, and uses their results to estimate the distribution of $|D^*(u_i)|$.

The second phase determines the frequentness of the remaining patterns, where each pattern S is processed by a task t_S so that the patterns are processed in parallel. If the mining loads of t_S is large, ScaleMine partitions it into subtasks either vertically (by domain table columns, recall Figure 2) or horizontally (i.e., by hashing data vertices among workers). Here, a task t_S is coarse-grained since it processes the entire domain table of a pattern; in contrast, T-FSM uses fine-grained tasks: each entry in a domain table is a subgraph-matching task. Also, a subtasks of T-FSM can run on any idle thread, while ScaleMine hardwires vertices to workers by hashing.

Moreover, evaluation in Phase 2 is approximate to save computation. We find that the results of ScaleMine can be different even for the same graph in different runs. The running time also tends to be very long when mining larger patterns ($\geq$ 6 vertices), likely due to using a suboptimal algorithm for subgraph matching.

DistGraph. DistGraph [27] partitions vertices to different workers so that the distributed memory can collectively hold a giant graph. DistGraph enumerates patterns in level-wise breadth-first search (BFS), where at level i it computes the support of candidate subgraph patterns comprising i edges. As a distributed system, it relies on efficient collective communication operations (AllToAll,

AllGather and AllReduce) to minimize communication, and uses pruning techniques to avoid communication for definitely (in)frequent patterns.

Since each graph partition is expanded by 1-hop in each round, the partitions can become very large after a few rounds and overlap a lot, leading to redundant computation. Moreover, each worker not only holds its partition but also the matched subgraph instances, leading to prohibitive memory space cost.

Fractal, Arabesque, RStream and Pangolin. These systems focus on unifying several graph mining problems such as motif counting and FSM. Their programming models materialize all the matched subgraph instances of the subgraph patterns, and count these instances to determine pattern frequentness. Arabesque [28], RStream [31] and Pangolin [4] expand the matched subgraph instances in BFS manner to create and examine larger and larger subgraph instances, which is very costly since the number of subgraph instances grows exponentially. This is in contrast to GraMi's early-termination idea that determines a pattern S as frequent as soon as its current support becomes larger than τ. Pangolin [4] exposes the pattern extending phase so that programmers can more effectively prune the enumeration space by eagerly detecting duplicate embeddings. Pangolin also allows architectural optimizations (e.g., data structures) and can run not only on CPU but also on GPU like cuTS [32].

Fractal [22] mitigates the performance issue by allowing its execution engine to conduct depth-first subgraph-instance backtracking without actually materializing the instances, but it still exhaustively mines all valid subgraph instances without any early termination (as in GraMi). Due to this algorithm inefficiency, Fractal requires users to specify a maximum pattern size n_{max}, so that patterns with more than n_{max} vertices will not be grown.

Peregrine. Peregrine [16] adopts a 'pattern-first' programming model that treats graph patterns as first-class constructs. This allows analysis of the pattern structure to more effectively guide the exploration on the data graph G. Peregrine's FSM program still grows patterns in a BFS manner, but instead of maintaining the huge number of intermediate pattern embeddings in G, Peregrine only maintains patterns themselves, and conducts subgraph matchings for their domain tables on demand to check pattern frequentness as in GraMi. Peregrine's subgraph matching algorithm for subgraph pattern S avoids non-canonical matches by enforcing a partial ordering on matched vertices to break pattern symmetries. It then computes the core of S as the subgraph induced by its minimum connected vertex cover, and computes matching orders by enumerating all permutations of vertices in the core that meet the partial ordering, which are then matched towards G.

As we shall see next, T-FSM uses the depth-first pattern extension method of gSpan that minimizes the number of active patterns that need subgraph matchings (and hence minimizes memory consumption). Moreover, T-FSM uses the latest subgraph matching algorithm [25] that does initial vertex candidate pruning and enumerates subgraphs in a small index (c.f. Figure 5). None of these optimizations is considered in Peregrine. Peregrine also treats the matching computation for each matching order as the smallest unit of parallelism, while T-FSM allows each subgraph matching task to further decompose if it times out, to avoid the straggler problem.

Pattern Enumeration. gSpan [36] is the SOTA algorithm for FSM in the transactional setting. It enumerates the subgraph patterns in depth-first manner. Figure 4 illustrates the pattern-growth tree of gSpan [36] where each circle represents a subgraph pattern, and each edge (S, S') grows pattern S by adding an adjacent edge to generate pattern S'. Different subgraph patterns may grow into the same pattern: for example, pattern A–B–C may be generated either by growing A–B by edge B–C, or by growing B–C by edge A–B. To avoid examining redundant patterns, gSpan encodes each subgraph pattern into a unique sequence called *DFS code* [36] computed from a 'DFS code tree' that connects pattern vertices in the order of their extension, and defines the minimum DFS

code among all isomorphic subgraphs of a pattern S as the canonical encoding of S, denoted by $min(S)$. For all isomorphic patterns, only the one whose DFS code is canonical would be processed. As an illustration, assume that S_a and S_b are isomorphic and since only S_a's encoding equals its canonical encoding, only S_a is checked for frequentness and for further pattern growth, while S_b (and its potential pattern-growth subtree) is pruned. Additionally, gSpan only extends a pattern S by an edge on the rightmost path of S's DFS code tree, since [36] shows that other extensions cannot be canonical.

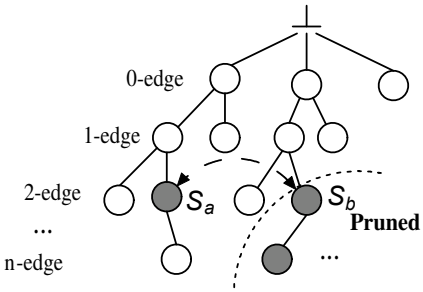

Fig. 4. Pattern-Growth Tree [36]

As an FSM algorithm in the transactional setting, gSpan [36] tracks the set of matched instances for each pattern S, so that they can be incrementally extended when considering patterns grown from S. In our single-graph setting, the pattern frequentness is examined using the domain table, so T-FSM only uses gSpan's depth-first pattern-growth scheme (with pattern canonicality check + rightmost path extension on DFS code tree) to enumerate new patterns for processing. This is more space-efficient than the BFS scheme of existing systems since we only keep a small number of active patterns for mining at any time: new patterns are evaluated only if space allows (i.e., some patterns finish evaluation).

Subgraph Matching. Subgraph matching finds all subgraph instances in a data graph G that is isomorphic to a given query graph S. Most existing SOTA algorithms for subgraph matching are built on top of Ullmann's backtracking algorithm [30]. Ullmann's algorithm orders the vertices of S as a sequence $\pi = [u_1, u_2, \ldots, u_k]$, and recursively matches each vertex u_i to one candidate data vertex in G with the same label. If u_k is matched, a pattern is found and outputted; while if u_i cannot find a match for some $i \leq k$, the algorithm backtracks to match u_{i-1} with its next candidate in G.

Sun and Luo [25] summarizes the SOTA algorithms on top of Ullmann's algorithm in terms of their optimization techniques, such as (1) initial candidate vertex pruning (e.g., v_9 in Figure 2 cannot be a candidate of u_2 since its degree is less than u_2's); (2) an auxiliary structure $\mathcal{A}_S$ for a pattern S, where each u_i is associated with a set of pruned candidates $C(u_i)$, and for each edge $(u_i, u_j) \in E^S$, its matched edges in G between $C(u_i)$ and $C(u_j)$ are materialized (to illustrate, Figure 5 shows $\mathcal{A}_S$ for pattern S in Figure 2); (3) query vertex ordering in π (e.g., u_i with smaller $|C(u_i)|$ should appear earlier in π to reduce recursion fanout). Ullmann's algorithm then enumerates the matched subgraphs on the smaller $\mathcal{A}_S$ rather than the original G following the selected vertex order π to achieve maximum efficiency. Following the recommendation by [25], we use the method of DP-iso [14] to compute $C(u_i)$ for all $u_i \in S$, use the method of GraphQL [15] to compute π, and use the DP-iso-style auxiliary structure.

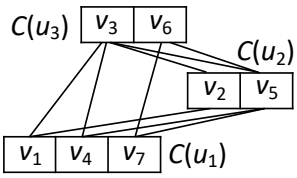

Fig. 5. Auxiliary Structure $\mathcal{A}_S$

As the SOTA serial FSM algorithm for a single graph, GraMi [10] simply treats subgraph matching as a constraint satisfaction problem (CSP), and [25] has shown that constraint programming is slower than Ullmann-style backtracking. In T-FSM, our subgraph-matching search stops as soon as a matched subgraph instance is found, but if there is no match, our search will complete the entire Ullmann-style backtracking algorithm, which may cause the straggler problem without our timeout-based task decomposition.

3 THE T-FSM SYSTEM

T-FSM is currently implemented as a shared-memory parallel system focusing on parallel task scheduling and computation, but it is easy to be extended for distributed execution using the vertex pulling technique of G-thinker [33], which we will discuss in Section 5. A T-FSM program runs a main thread called **worker**, and a pool of computing threads, called **compers**, for task computation.

This section first overviews the mining process of T-FSM and provides a brief cost analysis, followed by the technical details.

3.1 Overview of the Mining Process

Job Initialization and Initial Pattern Candidates. A T-FSM program begins by letting the worker (1) load the input graph G, (2) scan G to obtain and output the set of frequent vertex labels V_{freq} and frequent 1-edge patterns E_{freq}, (3) prune those edges of G that do not match any 1-edge pattern in E_{freq}, and (4) use E_{freq} to create an initial set of 2-edge candidate patterns, denoted by C^0_{pat}, for parallel task-based *pattern frequentness evaluation* and *pattern extension* by compers to maximize CPU utilization. To illustrate, recall Figure 2 and assume that support threshold $\tau = 3$. Then $V_{freq} = \{A, B\}$ and $E_{freq} = \{(A, B)\}$. Label C is pruned since only 2 vertices in G have label C, which is $< \tau$, and G is pruned to contain only 4 edges (v_1, v_2), (v_4, v_5), (v_5, v_7) and (v_8, v_9) to accelerate subsequent mining. Finally, $C^0_{pat} = \{A\text{–}B\text{–}A, B\text{–}A\text{–}B\}$.

Pattern Containers. Figure 6 overviews the system architecture of T-FSM with two pattern containers: (1) C_{pat} keeping candidate patterns to be evaluated, and (2) $\mathcal{L}_{active}$ keeping a list of active patterns currently under task-based frequentness evaluation.

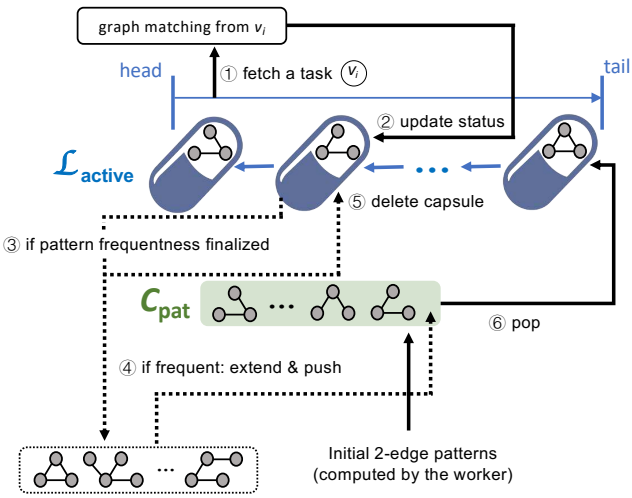

Fig. 6. System Architecture of T-FSM

Specifically, C_{pat} is a stack protected by a mutex for concurrent access by compers. We initialized C_{pat} with C^0_{pat}, the set of 2-edge candidate patterns obtained by the worker. When the capacity of $\mathcal{L}_{active}$ is not full, a comper may pop a new candidate pattern S from C_{pat} for evaluation, in which case S is added to $\mathcal{L}_{active}$ with its status (e.g., the domain table) being allocated and initialized. Also, when a frequent pattern is found, it will be extended with edges in E_{freq} to create more candidate patterns, which are then pushed to C_{pat}. We use gSpan's pattern extension approach which extends a new edge along the rightmost path of S's DFS code tree [36].

We implement C_{pat} as a stack so that we tend to grow existing frequent patterns to be larger to achieve a near depth-first traversal order on the pattern-growth tree (recall Figure 4), to keep the number of candidates in C_{pat} small (in contrast to queue-based BFS). Note that only subgraphs are kept in C_{pat} without any status information, so the memory consumption of C_{pat} is low.

On the other hand, $\mathcal{L}_{active}$ is a linked list of active patterns. To keep memory bounded, we only allow $\mathcal{L}_{active}$ to contain at most n^{max}_{active} active patterns, where n^{max}_{active} is a user-specified capacity parameter set to 32 by default. A pattern S appears earlier in L_{active}, if it was fetched from C_{pat} to

the tail of $\mathcal{L}_{active}$ for evaluation earlier. Whenever a comper tries to obtain a subgraph-matching task for processing, it will check the task availability of active patterns starting from the head of $\mathcal{L}_{active}$; the comper will fetch a task from the first active pattern $S \in \mathcal{L}_{active}$ with an available task. We implement $\mathcal{L}_{active}$ as such so that active patterns are evaluated in a near FIFO order to keep the number of active patterns minimal: a pattern that starts its evaluation earlier tends to be prioritized for task-based frequentness computation, so that its evaluation can finish sooner to make room for a new candidate pattern in C_{pat}.

As Figure 6 shows, each active pattern S in $\mathcal{L}_{active}$ is actually associated with a structure called "pattern capsule," which keeps the tasks of S and S's evaluation status. Section 3.2 will describe pattern capsule in detail, which is designed to (i) keep memory bounded by only having a small number of active subgraph-matching tasks at any time (as we shall see in Section 3.3), and to (ii) fully utilize pruning methods by keeping the latest pattern status.

Memory Cost Analysis. Recall that C_{pat} only keeps candidate subgraph patterns without any status information, so the memory cost is dominated by $\mathcal{L}_{active}$. The memory cost of $\mathcal{L}_{active}$ is well bounded since it can contain at most n_{active}^{max} active patterns at any time, and each active pattern is associated with a pattern capsule that contains a small and bounded number of active subgraph-matching tasks. Our appendix [19] provides an analysis and proof of our memory bound.

Worker Mining Procedure. In T-FSM, each comper is a thread that keeps fetching the next task for processing if available, or sets its state to idle otherwise. The worker periodically checks if there are still tasks to be processed (i.e., $\mathcal{L}_{active}$ or C_{pat} is not empty), or if some comper is still computing (which may generate new candidate patterns into C_{pat}). **Case i:** if so, it wakes up idle compers to process them (worker-compers notification is via condition variables); otherwise, **Case ii:** all compers are idle and $\mathcal{L}_{active}$ and C_{pat} are empty, so the worker terminates the T-FSM program.

Mining Procedure of a Comper: An Overview of the Steps. A comper keeps fetching tasks from $\mathcal{L}_{active}$ for evaluation as follows. It first scans $\mathcal{L}_{active}$ from the head to obtain a subgraph-matching task from the capsule of the first active pattern $S \in \mathcal{L}_{active}$ with an available task (c.f. ① in Figure 6). **Case 1:** if a task (for subgraph matching from v_i) is successfully fetched from the capsule of a pattern S, the task is executed and the status of S is updated accordingly (c.f. ②). **Case 1.1:** if this task determines that S is frequent (c.f. ③), it will extend S to generate larger candidate patterns, push them into C_{pat} (c.f. ④), and delete S from $\mathcal{L}_{active}$ (c.f. ⑤). While **Case 1.2:** if the task determines that S is infrequent (c.f. ③), it directly deletes S from $\mathcal{L}_{active}$ (c.f. ⑤). The comper then continues the next round to fetch another task from $\mathcal{L}_{active}$ for evaluation.

Case 2: if in a round, a comper cannot find any task after scanning the whole $\mathcal{L}_{active}$, then **Case 2.1:** if $|\mathcal{L}_{active}| < n_{active}^{max}$, the comper pops a new candidate pattern $S \in C_{pat}$, allocates a pattern capsule for S, and inserts it to the tail of $\mathcal{L}_{active}$ (c.f. ⑥); it then continues the next round to fetch another task from $\mathcal{L}_{active}$ for evaluation. While **Case 2.2:** if the capacity of $\mathcal{L}_{active}$ is full, the comper goes idle directly, which may be awakened by the worker later (i) to process new tasks, or (ii) to terminate the task probing loop if the worker flags the T-FSM program to terminate.

Time Cost Analysis. Recall that Step ④ in Figure 6 extends each frequent pattern by an edge using rightmost path extension and conducts canonicality check on each newly extended pattern as in gSpan, so no redundant patterns would ever be inserted in C_{pat}, and any pattern candidate in C_{pat} must be extended from a frequent pattern. In other words, we never examine patterns that are more than one-edge-extension away from the set of frequent patterns.

Let us denote the number of frequent patterns by n_{freq}, and assume that the average number of rightmost extensions of a frequent pattern's DFS code tree is n_{fanout}, then we examine at most $n_{freq} \cdot n_{fanout}$ candidate patterns (some may be filtered by canonicality check directly). While the value of n_{freq} and n_{fanout} is difficult to analyze (e.g., there is no such analysis in gSpan's

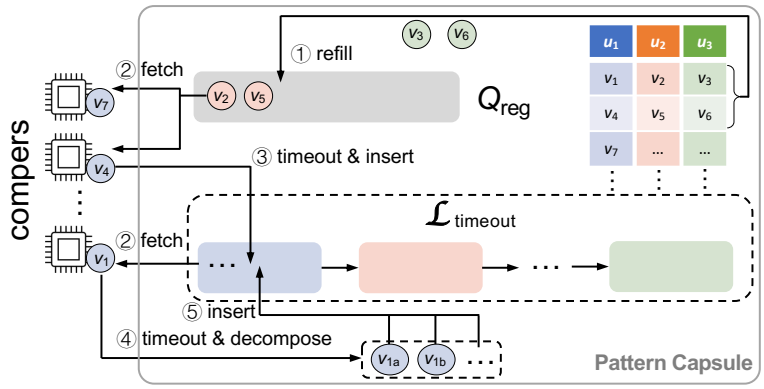

Fig. 8. Task Management Inside Pattern Capsule

paper [36]), their values are well-bounded in practice when a selective support threshold τ is used to find only very frequent patterns. For each candidate pattern S with vertices $\{u_1, u_2, \ldots, u_k\}$, each entry in the domain table may initiate a subgraph-matching task, giving at most $\sum_{i=1}^{k} |D(u_i)|$ subgraph-matching runs in total. Note that the actual number is much smaller since we use many pruning techniques (see Section 3.4). Let the average cost of each subgraph-matching run be C_{match}, then the time complexity of FSM is $O(n_{freq} \cdot n_{fanout} \cdot C_{match} \sum_{i=1}^{k} |D(u_i)|)$. While subgraph isomorphism is NP-complete [7], we use the latest algorithm for subgraph matching with a lot of pruning techniques [25] so C_{match} is well-bounded.

3.2 Task Containers in a Pattern Capsule

Recall that the frequentness evaluation of a pattern S with vertices $\{u_1, u_2, \ldots, u_k\}$ can be regarded as subgraph-matching tasks starting from individual data vertices $v \in D(u_i)$ $(i = 1, 2, \cdots, k)$ in the domain table of S (see Figure 2). Let us denote such a task by t_v.

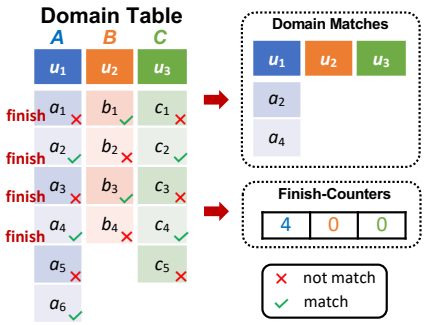

Fig. 7. Status Maintenance

Different subgraph-matching tasks can have drastically different computing workloads, so load balancing is essential to eliminate the straggler problem. In fact, we can avoid the full evaluation of some time-consuming tasks by postponing them after a timeout. For example, assume that our MNI support threshold is $\tau = 2$, and consider the evaluation of $D(u_1)$ in Figure 7. We will evaluate tasks t_{a_1}, t_{a_2}, t_{a_3} and t_{a_4} to find that u_1 does not breach the frequentness requirement, but if t_{a_1} and t_{a_3} are very time-consuming (e.g., need 10 seconds for full evaluation), a wiser solution is to put them aside after they run beyond a certain time threshold τ_{time} (we use $\tau_{time} = 0.1$ second by default), since t_{a_2} and t_{a_4} both confirm that a_2 and a_4 are valid matches to u_1 which suffices for $\tau = 2$. In this case, we can save 19.8 seconds! If after evaluating all non-timeout tasks in a domain $D(u_i)$, we still cannot find τ valid matches to u_i, we can then resume the evaluation of timeout tasks. For our previous example, if $\tau = 4$, then after evaluating the entire $D(u_1)$, we can find at most 3 matches, so we have to resume t_{a_1} and t_{a_3} to determine whether a_1 and a_3 are valid matches to u_1.

Figure 8 shows the internal of a pattern capsule for pattern S, where we create tasks t_v from entries v in the domain table of S into a regular task queue Q_{reg} (in gray) to be fetched by compers for subgraph matching. The tasks are created and appended to Q_{reg} following the order u_1, u_2, ...,

u_k. If a task times out, it will be temporarily moved to a timeout task list $\mathcal{L}_{timeout}$ (dashed rectangle). We organize $\mathcal{L}_{timeout}$ as a list of queues, one for each u_i to keep all the timeout tasks t_v where $v \in D(u_i)$. For $\mathcal{L}_{timeout}$ in Figure 8, the blue (resp. orange, green) queue keeps the timeout tasks of u_1 (resp. u_2, u_3). The queues of $\mathcal{L}_{timeout}$ are ordered based on the order of u_i's in S, so that compers will endeavor to finish all tasks from $D(u_i)$ before those from $D(u_j)$ if $i < j$. This is beneficial since if all tasks from $D(u_i)$ are finished and we find that $|D^*(u_i)| < \tau$, we know S is infrequent which avoids evaluating tasks from $D(u_j)$.

Task Fetching. Recall from ① in Figure 6 that a comper fetches a task t_v from some pattern capsule for subgraph matching at a time. We next explain how to decide where the comper fetches t_v from a pattern capsule, i.e., from Q_{reg} or from $\mathcal{L}_{timeout}$? We remark that all queues in Figure 8 are concurrent queues for efficient access by multiple compers. Additionally, $\mathcal{L}_{timeout}$ is protected by a read-write lock, where compers obtain tasks by reading the head of $\mathcal{L}_{timeout}$ for its first queue in most of the time, but when τ valid matches to u_i have been found, a comper will garbage-collect the queue for tasks of $D(u_i)$ from $\mathcal{L}_{timeout}$ after pattern status update.

We first define some concepts for ease of presentation. For a task t_v where $v \in D(u_i)$, we say that i is the **pattern-vertex ID (PID)** of t_v. In Figure 8, the PIDs of t_{v_2} and t_{v_5} in Q_{reg} are both 2 since $v_2, v_5 \in D(u_2)$. Also, let us denote the minimum PID of the tasks in Q_{reg} (resp. $\mathcal{L}_{timeout}$) by min_Q (resp. min_L). Since tasks are created from the domain table and appended to Q_{reg} in the strict order of $u_1, u_2, \ldots, u_k$, min_Q is the PID of the task at the head of Q_{reg}. If $Q_{reg} = \emptyset$ (resp. $\mathcal{L}_{timeout} = \emptyset$), we set $min_Q = \infty$ (resp. $min_L = \infty$).

Recall that a comper fetches a task t_v from some pattern capsule for subgraph matching at a time, and that we would like to finish all tasks from $D(u_i)$ before those from $D(u_j)$ if $i < j$. Following this principle, **Case 1:** if $min_Q > min_L$, a comper fetches the next task from $\mathcal{L}_{timeout}$. For example, in Figure 8 if the blue queue of $\mathcal{L}_{timeout}$ has tasks which are from $D(u_1)$, the comper should fetch a task from the queue rather than one from Q_{reg} which is from $D(u_2)$. **Case 2:** if $min_Q \leq min_L$ ($\triangleq i$), a comper fetches the next task from Q_{reg} since the evaluation of timeout tasks from $D(u_i)$ should be postponed till all tasks from $D(u_i)$ in Q_{reg} are processed.

A special case is when both Q_{reg} and $\mathcal{L}_{timeout}$ are empty, which we handle as follows. **Initially** when a pattern S is newly moved from C_{pat} to $\mathcal{L}_{active}$ (recall ⑥ in Figure 6), a comper will refill Q_{reg} using tasks from the domain table of S. **Later** when a comper finds that both Q_{reg} and $\mathcal{L}_{timeout}$ are empty in S's capsule, we determine that it fails to obtain a task from pattern S, so it will move to the next pattern in $\mathcal{L}_{active}$ (recall ① in Figure 6) to try to fetch a task; this is because a comper automatically refills Q_{reg} when Q_{reg} has insufficient tasks (see Section 3.3), so if Q_{reg} is empty, all entries from the domain table must have exhausted their tasks.

3.3 Task Processing by Each Comper

Assume that a comper successfully obtains a subgraph-matching task t_v from the current task capsule for processing as discussed above, Figure 8 shows the steps that the comper processes t_v.

Specifically, the comper first fetches t_v (c.f. ②). If t_v is from Q_{reg}, the comper first checks if Q_{reg} has less than n_{reg}^{min} tasks, where n_{reg}^{min} is a user-defined size lower bound for Q_{reg} (= 800 by default); if so, the comper refills up to n_{batch} tasks into Q_{reg} by creating them from entries in the domain table following the order of u_i's (c.f. ①). Here, n_{batch} is a user-defined batch size set to 800 by default. Note that Q_{reg} has at least n_{reg}^{min} tasks to keep compers busy, but Q_{reg} has at most $n_{reg}^{min} + n_{batch}$ tasks so the memory cost is bounded.

If a comper finishes t_v without timing out, it updates S's status (saved in the capsule) using the result, and then deletes t_v. Recall that if t_v times out, the comper will postpone its processing by adding it to $\mathcal{L}_{timeout}$ (c.f. ③ in Figure 8). However, if such a task t_v has to be finally evaluated, it can

Algorithm 1 *subgraph_match*(S, G)

```
1:  generate matching order π = [u_1, u_2, ..., u_k]
2:  enumerate(∅, 1, π, t_cur)
    Procedure enumerate(M, i, π, t_0):
3:    if i = k + 1 then output M; return
4:    C_M(u_i) ← viable vertex candidates in G to match u_i
5:    for each v ∈ C_M(u_i)
6:      append M with (u_i, v)
7:      if t_cur − t_0 ≤ τ_split do enumerate(M, i + 1, π, t_0)
8:      else create task ⟨M, i + 1, π⟩ and add it to system
9:      pop (u_i, v) from M
    Task ⟨M, i, π⟩:
10:   enumerate(M, i, π, t_cur)
```

still become a straggler if it is evaluated by only one comper. Therefore, we allow t_v (let $v \in D(u_i)$) to time out again, in which case we will decompose it into multiple smaller tasks (c.f. ④) which are added back to u_i's queue in $\mathcal{L}_{timeout}$ (c.f. ⑤) so that they can be processed by the compers in parallel (each may time out again and decompose).

We incrementally maintain min_Q for quick access. Specifically, whenever a comper fetches a task from Q_{reg} (c.f. ② in Figure 8) or refills tasks into Q_{reg} (c.f. ①), it updates min_Q to keep it up-to-date.

To determine min_L, a comper read-locks $L_{timeout}$ and probes it for the first non-empty queue. Assume that the queue is for keeping tasks from $D(u_i)$, then min_L is determined to be i.

Straggler Elimination by Task Decomposition. We next explain how we decompose a task t_v in Step ④ of Figure 8 when t_v times out. Algorithm 1 sketches our Ullmann-style recursive algorithm, where we match data vertices to query vertices $u_1, u_2, \ldots, u_k$ one at a time, with M recording the current partial match. The search process can be depicted by a search tree in Figure 9 for query graph S on data graph G in Figure 2. Here, for ease of presentation, we simply assume $C_M(u_i)$ in Lines 4–5 of Algorithm 1 to be equal to $C(u_i)$ as shown in Figure 5. The actual $C_M(u_i)$ in our implementation is much tighter, and please refer to Section 3.3 of [25] for the details of $C_M(u_i)$ computation.

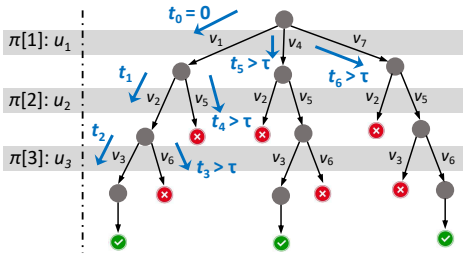

Fig. 9. Task Decomposition

In Figure 9, the leftmost path gives the recursion path $M = [(u_1, v_1), (u_2, v_2), (u_3, v_3)]$ which leads to a valid matching subgraph $\triangle v_1 v_2 v_3$ (see G in Figure 2). In contrast, path $M = [(u_1, v_1), (u_2, v_2), (u_3, v_6)]$ fails since $(v_2, v_6) \notin G$ so cannot match (u_2, u_3); in reality when using the auxiliary structure $\mathcal{A}_S$ shown in Figure 5, we will not extend $M = [(u_1, v_1), (u_2, v_2)]$ with (u_3, v_6), since $(v_2, v_6) \notin \mathcal{A}_S$.

In Algorithm 1, Line 2 implements a root task where $M = \emptyset$ and the task beginning time t_0 is set to be the current time t_{cur}. Lines 3–9 implements the Ullmann-style backtracking algorithm. Specifically, if k vertices have been successfully matched, Line 3 outputs the match M and returns. Otherwise, we proceed to match the next u_i whose candidate data vertices $C_M(u_i)$ are computed based on the current partial-match M: for each data vertex $v \in C_M(u_i)$, we match it to u_i by updating M in Line 6, and continue to match one more query vertex u_{i+1} by recursion in Line 7 (let us ignore timeout for now). Once the recursion returns, Line 9 removes that match (u_i, v) and continues to consider the next candidate in $C_M(u_i)$.

Next, let us consider how the timeout mechanism works to avoid straggler tasks. Note that after a task sets its initial time t_0, this initial time is passed on in subsequent recursion (c.f. Line 7). Each time before we match a candidate data vertex v to u_i, we check if more than τ_{split} time has passed since t_0 (c.f. Line 7, the if-condition). If not, we continue recursion; otherwise, the task times out and we create a new task $\langle M, i+1, \pi \rangle$ to process the corresponding search-space subtree rather than recursively processing it by the current task itself. The new task will reset its beginning time in Line 10 when it starts computation and may time out again in Line 7.

Figure 9 illustrates this timeout process: the current task traverses the search-space tree in depth-first order, processing for 3 recursive steps (t_0, t_1 and t_2) and when reaching Line 7 to check $v_6 \in C_M(u_3)$, the current time t_3 times out, so the subtree from extending v_6 is wrapped as an independent task by Line 8 and added to the system. Also, when the current task backtracks the recursion stack, three subsequent tasks are created due to timeout (t_4, t_5 and t_6). Note that these tasks are created at different granularities that are necessary and not over-decomposed.

So far, we described the general subgraph matching algorithm. In our application, each task t_v with $v \in D(u_i)$ starts by first matching u_i, so π is computed by fixing u_i as the first element, and computing the order among the remaining pattern vertices using the method described in Section 3.2 of [25]. In other words, each task t_v begins with $M = [(u_i, v)]$. Note that even though tasks from $D(u_i)$ of different pattern vertices u_i have different matching order π, the auxiliary structure $\mathcal{A}_S$ is the same, so $\mathcal{A}_S$ is created when we initialize the capsule of S and is used by all tasks for pattern S. Another difference of t_v from general subgraph matching is that, as soon as we find a complete match M, t_v completes by determining v as a valid match to u_i, rather than traversing the entire search-space tree to find all matches. For example, in Figure 9, t_v completes as soon as its finds the leftmost path to be a valid subgraph match.

CPU Utilization Analysis. Recall that we eliminate any straggler subgraph-matching task by decomposing it after running computation for τ_{split} time, and the decomposed tasks are inserted back to $\mathcal{L}_{timeout}$ for parallel processing by idle compers. Moreover, a comper automatically refills Q_{reg} by generating tasks from the domain table. As a result, both Q_{reg} and $\mathcal{L}_{timeout}$ have sufficient tasks to keep compers busy (unless tasks from domain table are exhausted for the current pattern S, in which case next pattern capsule in $\mathcal{L}_{active}$ will be accessed). Therefore, T-FSM can maintain a high CPU utilization rate during the course of FSM computation.

Capsule Deletion. Recall from Figure 6 that if a comper can determine that a pattern S is frequent or infrequent after processing a task (c.f. ③), it will delete the capsule of S from $\mathcal{L}_{active}$ (c.f. ⑤). However, this naïve solution may result in a segmentation fault.

Consider the pattern capsule in Figure 10 and assume that the support threshold $\tau = 2$. We also assume that (1) the pattern only has 3 entries yet to create subgraph-matching tasks: c_3, c_4 and c_5, while all other entries have finished (or pruned) their subgraph-matching tasks; and assume that (2) compers θ_1, θ_2 and θ_3 process tasks t_{c_3}, t_{c_4} and t_{c_5}, respectively. If t_{c_4} finds that c_4 is a valid match to u_3 so all u_i's have at least 2 valid matches, then S is determined to be frequent and θ_2 will delete S's capsule. However, θ_1 and θ_3 may still be processing

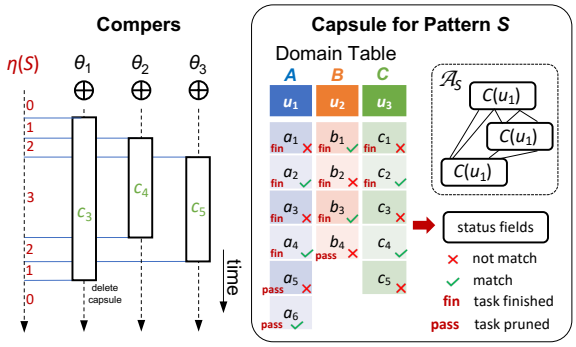

Fig. 10. Illustration of Capsule Deletion ($\tau = 2$)

Algorithm 2 An Iteration of Comper Computation

1: obtain the first capsule $S \in \mathcal{L}_{active}$ with available task(s)
2: lock $\eta(S)$; $\eta(S) \leftarrow \eta(S) + 1$; unlock $\eta(S)$
3: compute a task in capsule S and update its status (Fig. 6 ①②)
4: lock $\eta(S)$; $\eta(S) \leftarrow \eta(S) - 1$
5: **if** $\eta(S) = 0$ **and** S's frequentness is determined (Fig. 6 ③) **then**
6: extend S if it is frequent; delete S's capsule (Fig. 6 ④⑤)
7: **else** unlock $\eta(S)$

t_{c_3} and t_{c_5}, and will need to access $\mathcal{A}_S$ for subgraph enumeration and then update S's status in the capsule!

Our solution is to assign each pattern S a counter $\eta(S)$ that is kept in S's capsule, which records the number of compers that are currently processing subgraph-matching tasks of S or updating S's status. Algorithm 2 shows how we update the counter in each iteration of comper computation. As soon as we obtain a capsule $S \in \mathcal{L}_{active}$, we increment $\eta(S)$ in Line 2. After finishing a task $t_v \in S$, we decrement $\eta(S)$ in Line 4. Moreover, if the frequentness of S has been determined, and $\eta(S) = 0$ (i.e., the current comper θ is the last one still processing S), then θ deletes capsule S (including $\eta(S)$) in Line 6. Otherwise, $\eta(S)$ is unlocked in Line 7 so other compers can increment it again later in Line 2. In our previous example in Figure 10, θ_2 processing t_{c_4} will not delete S's capsule since $\eta(S) = 2$, and S's capsule will finally be deleted by θ_1 after finishing t_{c_3}. As an optimization, in Line 2 after a comper locks $\eta(S)$, if it finds that S's frequentness has been determined and $\eta(S) > 0$, it will unlock $\eta(S)$ and continue to check the next capsule in L_{active}.

In Algorithm 2, a comper needs to (a) read-lock $\mathcal{L}_{active}$ in Line 1, and (b) the lock can be released as soon as a task is obtained from S in Line 3 before conducting subgraph matching. If such a task cannot be obtained, (c) then the lock needs to be released right after Line 4, since (d) Line 6 needs to write-lock $\mathcal{L}_{active}$ to delete S's capsule. However, a deadlock can happen here. For example, consider two compers θ_1 and θ_2, where (1) θ_1 has read-locked $\mathcal{L}_{active}$ in Line 1 and is waiting to lock $\eta(S)$ in Line 2; while (2) θ_2 has locked $\eta(S)$ in Line 4 and is waiting to write-lock $\mathcal{L}_{active}$ in Line 6 to delete S's capsule. Our solution is to let Line 2 try-lock $\eta(S)$, so that if θ_1 failed to lock $\eta(S)$, it directly goes back to Line 1 to check the next capsule in $\mathcal{L}_{active}$; θ_1 will ultimately release the read-lock of $\mathcal{L}_{active}$, so θ_2 will be able to write-lock $\mathcal{L}_{active}$ to delete S's capsule.

3.4 Pattern Status and Pruning Techniques

Our MNI and Fraction-Score measures take **minimum** over the "frequencies" of matching data vertices for all $u_i \in S$. Recall from the end of Section 2.1 that $mni(S) = \min_{u_i \in S} |D^*(u_i)|$, and from Eq (9) and Eq (8) in Section 2.2 that $FS(S) = \min_{u_i \in S} \sigma_{u_i}(S) = \min_{u_i \in S} \sum_{v \in D^*(u_i)} \Delta^S(v)$. In other words, instead of adding 1 for each valid match $v \in D^*(u_i)$ as in MNI, $FS(S)$ adds $\Delta^S(v) = \min_{u_j \in N^S(u_i)} \Delta_{L^S(u_j)}(v)$ (c.f. Eq (5)) where $\Delta_\ell(v)$ is precomputed.

Lazy Search. Since we take minimum over the "frequencies" of matching data vertices for all $u_i \in S$, if some vertex $u_j \in S$ has frequency $|D^*(u_j)| < \tau$ (or $\sum_{v \in D^*(u_j)} \Delta^S(v) < \tau$), S is infrequent and there is no need to check the other u_i's. Similarly, if the current candidates of $u_j \in S$ has a cumulative frequency $\mathbf{n_j^F} \geq \tau$, then u_j will not be a reason for S to be infrequent, so the remaining candidates in $D(u_j)$ can be skipped. Let us denote the number of remaining candidates in $D(u_j)$ by $\mathbf{n_j^R}$. To summarize:

- If $n_i^F + n_i^R < \tau$, then S is flagged infrequent immediately.
- If $n_i^F \geq \tau$, then skip the remaining n_i^R tasks of u_i.

Here, we use the property that n_i^R upper-bounds the sum of $\Delta^S(v)$ for the remaining $v \in D(u_i)$, since $\Delta^S(v) \leq 1$ by Eq (2).

Note that each capsule keeps the status of its pattern S to facilitate pruning, which are a few flags, counters, and a Domain-Match Table (described below) that are lock-protected to be thread-safe. Here, we illustrate how these status fields assist lazy search.

- *Finish-Counter* (c.f. Figure 7) is an array where Finish-Counter$[i]$ represents how many tasks for u_i have finished. In Figure 7, suppose u_1's tasks a_1, a_2, a_3 and a_4 have finished, so Finish-Counter$[1] = 4$.
- *Domain-Match Table* (DMT, c.f. Figure 7) is a table where DMT$[i]$ keeps the set of valid matches in $D(u_i)$ found so far. For example, in Figure 2, when t_{a_1} finds a valid match $\triangle v_1 v_2 v_3$, it will insert v_3 to DMT$[3]$; later when another match $\triangle v_4 v_5 v_3$ is found by another task, v_3 will be inserted again so DMT$[3]$ is organized as a set for deduplication. In Figure 7, while tasks a_1, a_2, a_3 and a_4 have finished, only a_2 and a_4 are valid matches, so they are in DMT$[1]$.

With Finish-Counter and DMT, lazy search is straightforward. For example, in Figure 7, if $\tau = 5$, as soon as a_3 is finished, there is no need to examine u_1's remaining tasks and all tasks of u_2 and u_3: $n_1^F = |\text{DMT}[1]| = 1$ (i.e., a_2 is found valid), and $n_1^R = |D(u_1)| - \text{Finish-Counter}[1] = 6 - 3 = 3$, so $n_1^F + n_1^R < \tau$ and the whole pattern is flagged infrequent immediately.

Domain Initialization. Assume that a pattern S is obtained by extending an edge from its parent pattern S_{pa}. When creating S's capsule, we initialize the domain table of S with that of S_{pa}. This is because for any $u_i \in S_{pa}$, if $v \in D(u_i)$ is invalid in S_{pa}, it must also be invalid for S so can be pruned. This conclusion holds generally for any subgraph S' of S, besides S_{pa}. Therefore, GraMi [10] caches all patterns that are examined so far, which are indexed by signatures that allows to quickly identify those subgraph patterns of S with one edge removed (denoted by S'), and GraMi removes from $D(u_i)$ of S those invalid candidate vertices $v \in D(u_i)$ previously found for S'. This **push-down pruning** technique is inherited by T-FSM.

To implement this parent-inherited domain table initialization, in Figure 6, when Step ⑤ deletes the capsule of a frequent pattern S_{pa}, its domain table (denoted by $\mathcal{T}_{pa}$) will not be deleted immediately, so that its extended child-patterns S can access $\mathcal{T}_{pa}$ to create their own domain tables. To ensure that $\mathcal{T}_{pa}$ will be properly garbage-collected, we maintain a counter η_{pa} with $\mathcal{T}_{pa}$ which gets incremented whenever a child pattern is fetched for processing (c.f. ⑥). Assume that S_{pa} was extended to create n_{pa} child patterns, then the last child pattern will find $\eta_{pa} = n_{pa}$ so it will delete $\mathcal{T}_{pa}$.

Other Pruning Rules. T-FSM integrates all pruning rules of GraMi. Besides *push-down pruning* above, it also uses the following 3 pruning rules. (1) **Unique label:** [10] proves that if pattern S is acyclic and every $u_i \in S$ has a distinct label, then $D(u_i) = D^*(u_i)$ so subgraph matching is not needed. As a result, if S satisfies this requirement, a comper directly validates S's frequency without moving S to $\mathcal{L}_{active}$ by creating a capsule. (2) **Decomposition pruning:** if a task t_v of S timed out and was moved to $\mathcal{L}_{timeout}$ of a capsule, the timeout tasks of S tend to be expensive so we conduct additional pruning of S by removing an edge from S (let the subgraph be S^-), computing connected components (CCs) of S^-, and check if subgraph matching from v using any CC fails; if so, t_v fails immediately without using S for subgraph matching. (3) **Automorphism:** it avoids redundant computation for symmetric vertices in a pattern. Specifically, if u_i and u_j are symmetric and u_i has a valid match v, then v must also be a valid match for u_j.

Lineage Tracking of Timeout Tasks. Recall from Line 8 of Algorithm 1 that a task t_v may time out and be decomposed into many smaller tasks, each of which may again decompose, forming a task lineage tree with a tree edge (t_{pa}, t_{cur}) if t_{cur} is created by decomposing t_{pa} (which generates m tasks). This lineage tree is tracked so that whenever t_{cur} completes, it will increment a counter

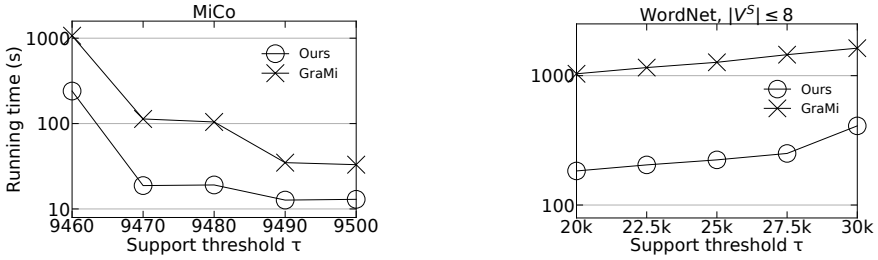

Fig. 11. Our Implementation v.s. Original GraMi [12]

at t_{pa}, and if the counter equals m, we increment the counter of t_{pa}'s parent task; this process goes on, and if the counter of task t_v at the tree root indicates that all t_v's child tasks are complete, then t_{cur} is the last task of t_v, so it updates S's status to flag v as valid match to u_i iff any task in the task lineage tree found a valid match. In fact, as soon as a task t_{cur} finds a valid match, it will flag v as valid match to u_i, and the other subtasks of t_v will probe this flag and terminate immediately (we let each task probe pattern status at Line 8 of Algorithm 1 so that subtasks will not be generated if v is already flagged as valid or u_i is already flagged as (in)frequent).

Timeout tasks may cause another problem. For example, if tasks t_{a_2} and t_{a_4} in Figure 7 finish so two valid matches a_2 and a_4 are found, then if $\tau = 2$, we can flag u_1 as frequent; the subtasks of t_{a_3} will probe and detect this flag to skip their computation, but since t_{a_3} is not complete, we do not know whether a_3 is a valid match, so we should not flag a_3 as invalid. To implement this, when t_{cur} skips computation, we let it propagate the SKIP flag upwards in the lineage tree; if t_v at the tree root gets a SKIP flag, $v \in D(u_i)$ should not be flagged invalid.

4 EXPERIMENTS

We now evaluate T-FSM and compare it with SOTA systems ScaleMine [1], Fractal [22], DistGraph [27], Pangolin [4] and Peregrine [16]. These systems only support MNI so we use MNI by default. Section 4.7 compares MNI with Fraction-Score using T-FSM.

Table 2. Datasets

Dataset	|V|	|E|	d_{avg}	|L|	Category
GSE1730	998	5,096	10.21	2	Biology
Yeast*	3,112	12,519	8.05	71	Biology
Human*	4,674	86,282	36.92	44	Biology
WordNet*	76,853	120,399	3.13	5	Lexical
MiCo*	100,000	1,080,298	21.61	29	Citation
UK POI*	182,334	2,816,000	30.89	36	Spatial
DBLP	317,080	1,049,866	6.62	15	Social
Youtube	1,134,890	2,987,624	5.27	25	Social
Patent*	2,745,761	13,965,409	10.17	37	Citation
Twitter	11,316,811	85,331,846	15.08	25	Social

4.1 Datasets and Environmental Setup

We select 10 real-world graph datasets with various sizes, densities and categories as summarized in Table 2, where datasets *Yeast*, *Human*, *WordNet*, *DBLP* and *Youtube* are from [25], *GSE1730* is from [13], *MiCo* is from [10], *Patent* is from [1], *UK POI* is from [3] where an edge is added between every pair of POIs with distance $\leq$ 500 m, and *Twitter* is from [29]. Datasets with name accompanied by "*" come with their original vertex labels, and *MiCo* additionally has edge labels. The remaining graphs are not labeled, so we follow previous works [25, 26] by randomly assigning a label to each vertex from a predefined label set L. Experiments are run on a server with 48 cores (Intel Xeon Gold 6248R CPU 3.00GHz) and 64 GB RAM. Each reported experiment is repeated for 3 times with the average reported. We use $\tau_{time} = 0.1$ s and $\tau_{split} = 1$ s for T-FSM by default.

4.2 Serial GraMi Implementation

We notice that the original implementation of GraMi at [12] has several performance issues: (1) instead of using effective heuristics to find a good matching order $[u_1, u_2, \ldots, u_k]$, GraMi uses a

simple DFS order which might not be optimal as indicated in [25]; (2) GraMi does not utilize an auxiliary structure $\mathcal{A}_S$ but enumerates directly on G. We, therefore, integrate the latest subgraph matching algorithm with GraMi's logic to develop a more efficient C++ implementation. We compare our serial implementation with that of [12]. Figure 11 illustrates their performance on *MiCo* and *WordNet*, which shows that our implementation is 3× to 5× faster. Note that since large patterns on *WordNet* are expensive to mine, we stop extending a pattern if its size reaches 8 vertices (i.e., $|V^S| \leq 8$). The time increases with τ on *WordNet* since more candidates in each $D(u_i)$ needs evaluation, but the # of results does not change much.

4.3 Comparison with Existing Systems

Figure 12 shows the performance of T-FSM compared with ScaleMine [23], Fractal [11], DistGraph [9], Pangolin [20] and Peregrine [21] on all 10 graphs shown in Table 2 for different values of support threshold τ. For some graphs, mining large patterns are expensive, so following [1], we stop extending a pattern if its size goes beyond a certain threshold. We also set the maximum running time as 10^4 seconds. In Figure 12, "M" means Out of Memory (64 GB), "T" means Out of Time (10^4 s), "A" means the program aborts (occurs only in Pangolin [20] due to an assertion error), and "X" means results returned are not exact. Note that even though a program fails and may terminate quickly for the case of "M" or "A," we still plot its hatched bar to the top like for "T" to help readers easily see which system the bar corresponds to. Also, there is no bar on *MiCo* for Fractal and Peregrine since they do not support edge labels.

Figure 12 shows that T-FSM generally has the best performance on all the graphs, especially the denser ones. For example, on *Human*, T-FSM is the only system that can mine all pattern with no more than 6 vertices, since *Human* is very dense with an average degree of 36.92. A similar observation can be reached for *UK POI*, where ScaleMine is the only other system that can finish for some tested values of τ.

On *GSE1730* and *DBLP*, although ScaleMine sometimes has a lower running time than T-FSM, we observe that ScaleMine's results are frequently not exact and even inconsistent from two different runs. For example, on *DBLP* when $\tau = 1800$, T-FSM returns 1745 patterns which is the same as that returned by GraMi [12], but ScaleMine returns 830 patterns in one run and 832 patterns in another run. This shows that the approximate pruning techniques of ScaleMine can be far from accurate.

DistGraph can only handle two graphs *DBLP* and *Patent*. Since DistGraph extends patterns by BFS, it suffers frequently from Out-of-Memory errors on *Human*, *MiCo*, *UK POI*, *Youtube* and *Twitter*. The approach does work well when the number of patterns is small, where performance comparable to T-FSM is achieved on *DBLP* when $\tau = 1800$ and 2000. The problem of pattern extension by BFS also applies to Pangolin, which aborts on all datasets except for *DBLP* and *Patent*. Moreover, we find that the results of Pangolin are different from the exact results on *DBLP* (see Section 4.4), which is likely due to implementation issues.

Although Fractal supports depth-first pattern extension, it exhaustively mines all valid subgraphs without any early termination (after τ matches are found), so it is the slowest among all systems. Peregrine conducts breadth-first pattern extension but considers domain table as in T-FSM where subgraph matching is conducted in a depth-first manner. However, its subgraph matching and load balancing approaches are less efficient. As a result, Peregrine runs out of time (10^4 s) on most datasets except for *DBLP* and *Patent*, and *Youtube* only when $\tau = 2000$. Both Fractal and Peregrine do not support edge labels, so we cannot show their results on *MiCo*.

We show the CPU utilization rates of all the systems when running 32 computing threads on *Patent* with $\tau = 24k$, in which setting all systems can complete their FSM jobs. Figure 13 shows the CPU rates for the first 120 seconds, where we can see that the CPU rate of T-FSM (blue line) jumps to 3200% and completes the job quickly. DistGraph and Peregrine also achieve 3200% but their

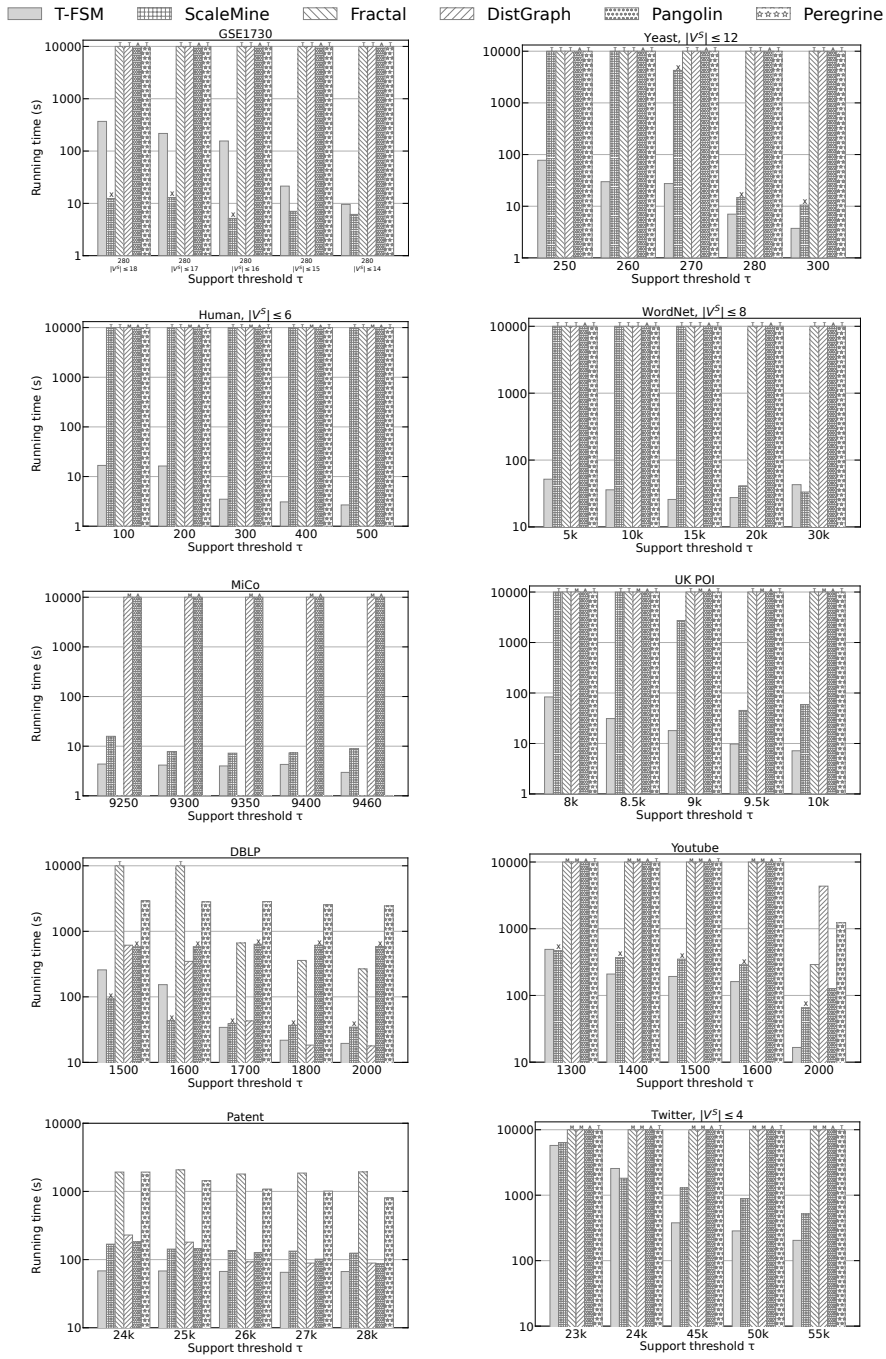

Fig. 12. T-FSM v.s. existing FSM systems with 32 Compers

running times are much longer. ScaleMine has a slightly lower CPU rate but is able to complete within 120 seconds. The peak CPU rate of Fractal is far from 3200%, while the CPU rate of Pangolin gradually drops with a very long tailing period of low CPU utilization rate.

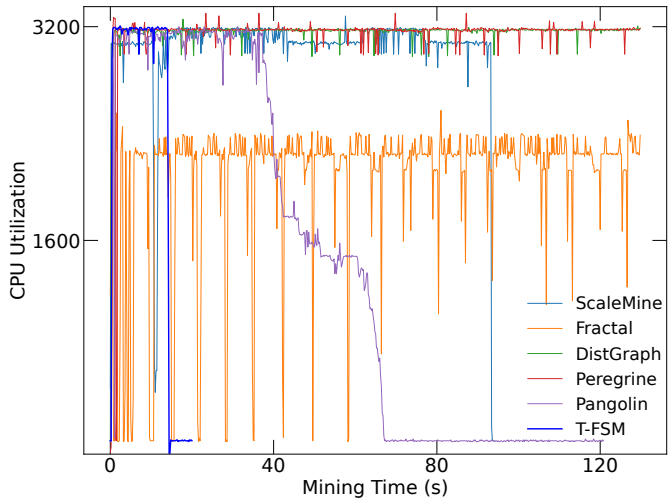

Fig. 13. CPU Utilization Rates

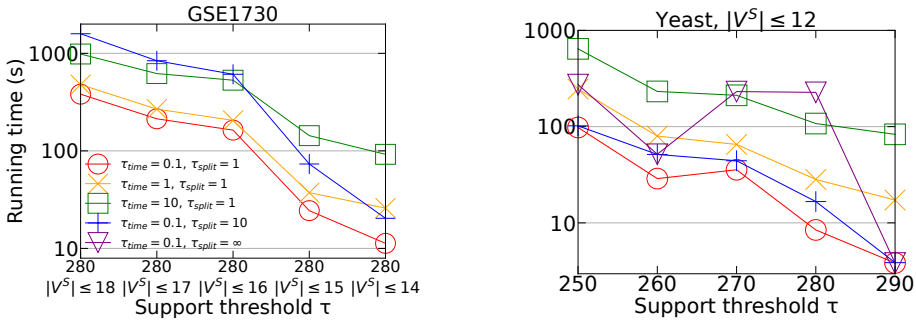

Fig. 14. Ablation Study

4.4 Result Exactness of ScaleMine and Pangolin

Recall that ScaleMine returns only approximate results. Table 3 shows the exact number of subgraph patterns (i.e., MNI) and those returned by ScaleMine on *Youtube* and *DBLP*, where we see that ScaleMine misses a lot of patterns due to an overly aggressive pruning heuristic. Surprisingly, we find that Pangolin returns more patterns than the exact ones on *DBLP*, even though it is supposed to be an exact approach. This is likely due to implementation issues.

Table 3. Number of Results

Youtube

Support	2,000	1600	1500	1400	1300
MNI	350	6678	7846	8296	14,577
ScaleMine	332	2741	3224	3283	3614

DBLP

Support	2000	1800	1700	1600	1500
MNI	1695	1745	2826	15,347	25,842
ScaleMine	785	830	997	1182	2411
Pangolin	3047	3178	4202	15,986	26,065

4.5 Ablation Study

Recall that when evaluating candidates of $D(u_i)$, we put a candidate v aside into $\mathcal{L}_{timeout}$ if t_v runs for more than $\tau_{time} = 0.1$ s; and if a task from $\mathcal{L}_{timeout}$ runs for more than $\tau_{split} = 1$ s, we decompose it again to avoid the straggler problem. These default parameter values have been carefully tuned to work well in general.

There are performance tradeoffs for the values of τ_{time} and τ_{split}. Specifically, if there are sufficient candidates $v \in D(u_i)$ that can be quickly checked to be a valid match to u_i, then if τ_{time} is set too

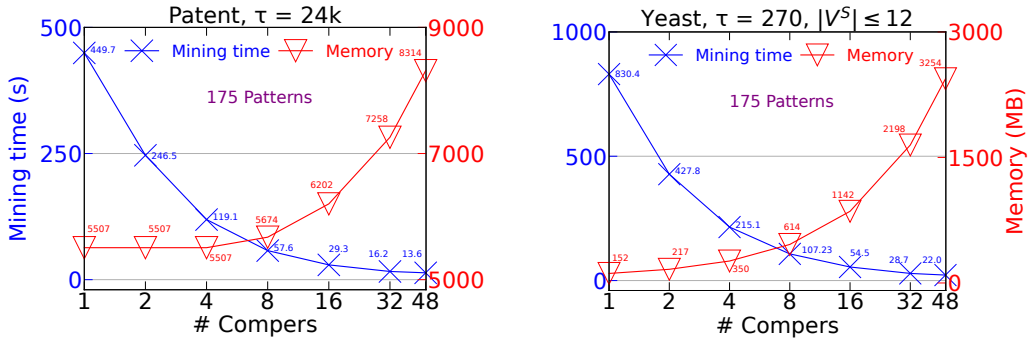

Fig. 15. Scalability

large, we might stick on expensive tasks t_v that can otherwise be skipped to save time. Also, while a smaller τ_{split} improves load balancing, it causes more overhead for subtask creation and scheduling. We tested different combinations of τ_{time} and τ_{split}, and some results are shown in Figure 14 for *GSE1730* and *Yeast*. Note that our default parameters (red line) lead to the best performance.

Moreover, these techniques are very important in reducing the running time. For example, if we increase τ_{time} from 0.1 s to 10 s (green line), the running time is increased by up to 8× on *GSE1730* and 20× on *Yeast*. We did not further increase τ_{time} in Figure 14 since the time would continue to increase significantly. Also, if we increase τ_{split} from 1 s to ∞ (purple line), the running time is increased by up to 30× on *Yeast*. We did not show the results for $\tau_{split} = \infty$ on *GSE1730* in Figure 14 since the job runs out of time.

4.6 Scalability

Figure 15 shows the vertical scalability of T-FSM, i.e., the running time and peak memory consumption when we vary the number of compers as 1, 2, 4, 8, 16, 32 and 48 on *Patent* and *Yeast*. We can see that the scale-up speedup is near ideal: nearly 27× (resp. 30×) on *Patent* (resp. Yeast) with 32 compers, nearly 33× (resp. 37×) on *Patent* (resp. Yeast) with 48 compers. Note that even with 48 compers, the memory space consumed by tasks is only around 3 GB. *Patent* uses more memory because its graph is much larger.

4.7 Fraction-Score v.s. MNI

Accuracy Comparison. To study the support accuracy of Fraction-Score and MNI, we generate a synthetic dataset following the exact approach in Section IV-A of [3] so that we can obtain the ground-truth support for comparison.

We configure the data-generation parameters defined in [3], N_{co_loc}, $m_{overlap}$, λ_1, λ_2, D, d, m_{clump}, r_{noisy_label} and r_{noisy_num}, to be 20, 10, 5, 40, 10^6, 5, 5, 0.5 and 0.5, respectively. We refer interested readers to [3] for the details, but give a brief but intuitive description here. Specifically, we consider a spatial area of size $D \times D$ that is partitioned by a grid with cell size $d \times d$. Since $D = 10^6$ and $d = 5$, we have $200,000 \times 200,000$ grid cells in total. The goal is to generate a labeled object dataset that contains $N_{co\_loc} = 20$ frequent co-location patterns. For each frequent pattern, say $S = \{A, B, D, E, F\}$ shown in Figure 16, we put the corresponding data instances into n_S grid cells, where n_S is randomly sampled from a Poisson distribution with mean λ_2. In Figure 16, we have $n_S = 3$. For each cell, we randomly generate n_ℓ data instances for each label $\ell \in S$, where n_ℓ is a random integer sampled from 1 to $m_{clump} = 5$. In Figure 16, we have $n_A = 3$, $n_B = 2$, $n_D = 2$, $n_E = 3$ and $n_F = 2$ for all the cells. Some noisy data instances and noisy labels are also added following [3]

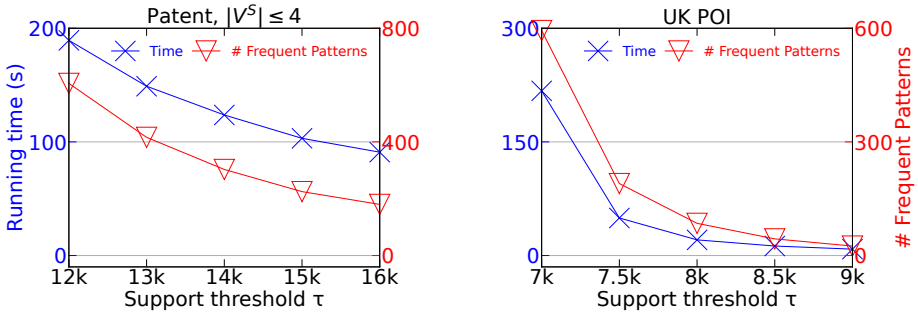

Fig. 18. Results with Fraction-Score as Support

but they usually do not produce frequent patterns. In total, our dataset has 359,262 objects and 441 labels.

The ground-truth support of each frequent co-location pattern S can be easily estimated. For example, in Figure 16, each of the 3 cells has the smallest m_{clump} value being 2 (i.e., for labels B, D and F), so we assume it contributes 2 to the overall support assuming that every pair of data instances in the $d \times d$ cell are within distance d (which happens with a high probability). We also assume that instances from different cells will not form a match to S, which is usually true since we have many grid cells but λ_2 is merely 40. Therefore, the total support is obtained by summing the support from all occupied cells, which is $2 + 2 + 2 = 6$ for co-location pattern $\{A, B, D, E, F\}$ in Figure 16. We create a graph from this dataset for FSM, by indexing all objects in an R-tree and conducting a range query of radius $d = 5$ on each object to build its adjacency list.

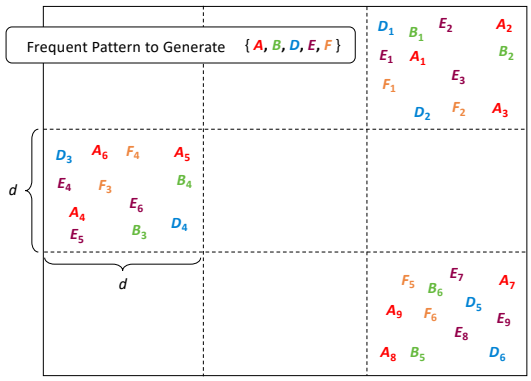

Fig. 16. Synthetic Dataset Illustration

Figure 17 shows the support of the top-10 most frequent ground-truth patterns, as well as their MNI and Fraction-Score. We can see that while both measures overestimates the ground-truth support, Fraction-Score is much closer so is clearly a better measure.

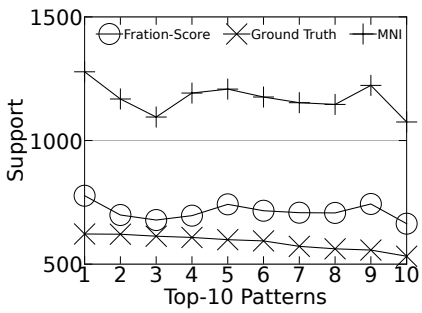

Fig. 17. Support Values

Running Time and Pattern Number. So far, we have only reported T-FSM efficiency when MNI is used as the support. The results for Fraction-Score are similar. To illustrate, Figure 18 shows the running time and the number of frequent patterns on *Patent* and *UK POI* using Fraction-Score as the support when τ varies, and for comparison purpose, Figure 19 shows these results using MNI as the support. As expected, the running time and pattern number decrease as τ increases. However, the running time (resp. pattern number) of Fraction-Score is much shorter (resp. smaller) than MNI for the same τ. This is because Fraction-Score is a much tighter support measure than MNI, so most overestimated false-positive patterns are avoided. Note that we add a restriction $|V^S| \leq 4$ on *Patent* since otherwise the running time using MNI is too long.

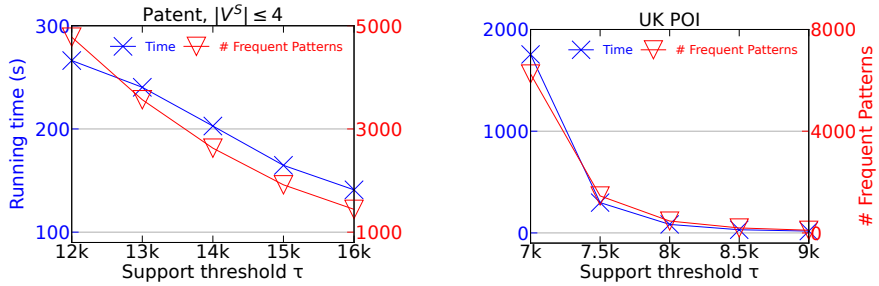

Fig. 19. Results with MNI as Support

Pattern Visualization. Figure 20 shows two example frequent patterns from the POI dataset *UK POI* that are not co-location patterns, where each vertex represents a POI type. For example, "Community" POIs locate in residential areas (e.g., churches). In Figure 20, we indicate the frequency of POI types in red under the vertices, and we indicate the Fraction-Score of each pattern in blue.

The first pattern (left) contains four POI types, and has a Fraction-Score of 4245.71 which is very high considering that there are only 9390 "Supermarkets" POIs in *UK POI*. The pattern matches our intuition that a convenient residential area usually has "Supermarkets" and "Food" POIs nearby, both of which have an ATM POI within a reachable range for customers to withdraw cash. However, the "Supermarkets" POI and "Food" POI do not have to be very close to each other. The second pattern (right) is larger with 5 POI types, so its Fraction-Score (1031.28) is smaller. Again, we see the "Community"–"Supermarkets"–"Food" triangle, as well as another "Supermarkets"–"Car_Parts"–"Fuel" triangle which matches intuition (e.g., a Costco store usually co-locates with a "Tire Center" and a gas station).

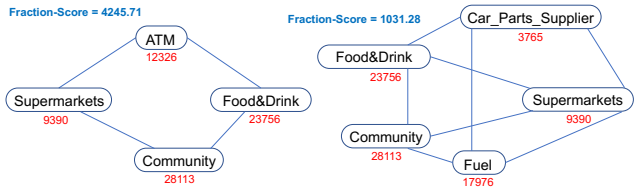

Fig. 20. Two Frequent Patterns from *UK POI*

We can see that our FSM formulation finds much richer POI patterns than conventional co-location pattern mining formulation.

5 CONCLUSIONS AND FUTURE WORK

We presented an efficient system called T-FSM for parallel mining of frequent subgraph patterns in a big graph. T-FSM adopts a novel task-based execution engine design to ensure high concurrency, bounded memory consumption, effective load balancing. T-FSM integrates the latest subgraph matching algorithm to enable the efficient mining of much larger patterns. It also supports a new measure called Fraction-Score which is more accurate than the widely used MNI measure. Our experiments show that T-FSM is orders of magnitude faster than SOTA systems for FSM.

While we have assumed the input graph G to be undirected in our description, our approach can be easily generalized to mine a directed graph, by treating edge direction as part of an edge label. For example, assume we grow a pattern S by an edge from a vertex $u \in S$: $u \xrightarrow{A} v$ (resp. $u \xleftarrow{A} v$), then we treat the edge as undirected but with the edge label being a tuple $\langle A, out \rangle$ (resp. $\langle A, in \rangle$).

As a future work, we will extend T-FSM to a shared-nothing distributed environment. This can be achieved by maintaining a distributed vertex store as in G-thinker (c.f. Section V-A of [33]) for collecting task data, and by using the task management design of G-thinker (c.f. Section V-B of [33]) to allow tasks to wait for data and to be notified to compute when its data are ready.

REFERENCES

[1] Ehab Abdelhamid et al. "Scalemine: scalable parallel frequent subgraph mining in a single large graph". In: *SC*. 2016, pp. 716–727.

[2] Björn Bringmann and Siegfried Nijssen. "What Is Frequent in a Single Graph?" In: *PAKDD*. Vol. 5012. Lecture Notes in Computer Science. Springer, 2008, pp. 858–863.

[3] Harry Kai-Ho Chan et al. "Fraction-Score: A New Support Measure for Co-location Pattern Mining". In: *ICDE*. IEEE, 2019, pp. 1514–1525.

[4] Xuhao Chen et al. "Pangolin: An Efficient and Flexible Graph Mining System on CPU and GPU". In: *Proc. VLDB Endow.* 13.8 (2020), pp. 1190–1205.

[5] Young-Rae Cho and Aidong Zhang. "Predicting protein function by frequent functional association pattern mining in protein interaction networks". In: *IEEE Trans. Inf. Technol. Biomed.* 14.1 (2010), pp. 30–36.

[6] Wei-Ta Chu and Ming-Hung Tsai. "Visual pattern discovery for architecture image classification and product image search". In: *International Conference on Multimedia Retrieval, ICMR*. ACM, 2012, p. 27.

[7] Stephen A. Cook. "The Complexity of Theorem-Proving Procedures". In: *STOC*. ACM, 1971, pp. 151–158.

[8] Mukund Deshpande et al. "Frequent Substructure-Based Approaches for Classifying Chemical Compounds". In: *IEEE Trans. Knowl. Data Eng.* 17.8 (2005), pp. 1036–1050.

[9] *DistGraph*. https://github.com/zakimjz/DistGraph.

[10] Mohammed Elseidy et al. "GRAMI: Frequent Subgraph and Pattern Mining in a Single Large Graph". In: *Proc. VLDB Endow.* 7.7 (2014), pp. 517–528.

[11] *Fractal*. https://github.com/dccspeed/fractal.

[12] *GraMi*. https://github.com/ehab-abdelhamid/GraMi.

[13] *GSE1730*. https://www.ncbi.nlm.nih.gov/geo/query/acc.cgi?acc=GSE1730.

[14] Myoungji Han et al. "Efficient Subgraph Matching: Harmonizing Dynamic Programming, Adaptive Matching Order, and Failing Set Together". In: *SIGMOD*. ACM, 2019, pp. 1429–1446.

[15] Huahai He and Ambuj K. Singh. "Graphs-at-a-time: query language and access methods for graph databases". In: *SIGMOD*. ACM, 2008, pp. 405–418.

[16] Kasra Jamshidi, Rakesh Mahadasa, and Keval Vora. "Peregrine: a pattern-aware graph mining system". In: *EuroSys*. ACM, 2020, 13:1–13:16.

[17] Amine Mhedhbi and Semih Salihoglu. "Optimizing Subgraph Queries by Combining Binary and Worst-Case Optimal Joins". In: *Proc. VLDB Endow.* 12.11 (2019), pp. 1692–1704.

[18] Siegfried Nijssen and Joost N. Kok. "The Gaston Tool for Frequent Subgraph Mining". In: *Electron. Notes Theor. Comput. Sci.* 127.1 (2005), pp. 77–87.

[19] *Online Appendix*. https://github.com/lyuheng/T-FSM/blob/main/appendix.pdf.

[20] *Pangolin*. https://github.com/chenxuhao/GraphMiner.

[21] *Peregrine*. https://github.com/pdclab/peregrine.

[22] Vinicius Vitor dos Santos Dias et al. "Fractal: A General-Purpose Graph Pattern Mining System". In: *SIGMOD*. ACM, 2019, pp. 1357–1374.

[23] *ScaleMine*. https://github.com/ehab-abdelhamid/ScaleMine.

[24] Madeleine Seeland et al. "Online Structural Graph Clustering Using Frequent Subgraph Mining". In: *ECML PKDD*. Vol. 6323. Lecture Notes in Computer Science. Springer, 2010, pp. 213–228.

[25] Shixuan Sun and Qiong Luo. "In-Memory Subgraph Matching: An In-depth Study". In: *SIGMOD*. 2020, pp. 1083–1098.

[26] Shixuan Sun et al. “RapidMatch: A Holistic Approach to Subgraph Query Processing”. In: *Proc. VLDB Endow.* 14.2 (2020), pp. 176–188.
[27] Nilothpal Talukder and Mohammed J. Zaki. “A distributed approach for graph mining in massive networks”. In: *Data Min. Knowl. Discov.* 30.5 (2016), pp. 1024–1052.
[28] Carlos H. C. Teixeira et al. “Arabesque: a system for distributed graph mining”. In: *SOSP*. ACM, 2015, pp. 425–440.
[29] *Twitter*. https://academictorrents.com/details/2399616d26eeb4ae9ac3d05c7fdd98958299efa9.
[30] Julian R. Ullmann. “An Algorithm for Subgraph Isomorphism”. In: *J. ACM* 23.1 (1976), pp. 31–42.
[31] Kai Wang et al. “RStream: Marrying Relational Algebra with Streaming for Efficient Graph Mining on a Single Machine”. In: *OSDI*. USENIX Association, 2018, pp. 763–782.
[32] Lizhi Xiang et al. “cuTS: scaling subgraph isomorphism on distributed multi-GPU systems using trie based data structure”. In: *SC*. ACM, 2021, 69:1–69:14.
[33] Da Yan et al. “G-thinker: A Distributed Framework for Mining Subgraphs in a Big Graph”. In: *ICDE*. IEEE, 2020, pp. 1369–1380.
[34] Da Yan et al. “PrefixFPM: A Parallel Framework for General-Purpose Frequent Pattern Mining”. In: *ICDE*. IEEE, 2020, pp. 1938–1941.
[35] Da Yan et al. “PrefixFPM: a parallel framework for general-purpose mining of frequent and closed patterns”. In: *VLDB J.* 31.2 (2022), pp. 253–286.
[36] Xifeng Yan and Jiawei Han. “gSpan: Graph-Based Substructure Pattern Mining”. In: *ICDM*. 2002, pp. 721–724.
[37] Xifeng Yan, Philip S. Yu, and Jiawei Han. “Graph Indexing: A Frequent Structure-based Approach”. In: *SIGMOD*. ACM, 2004, pp. 335–346.
[38] Zongliang Yue et al. “Biological Network Mining”. In: *Modeling Transcriptional Regulation*. Springer, 2021, pp. 139–151.
[39] Lei Zou, Lei Chen, and M. Tamer Özsu. “K-Automorphism: A General Framework For Privacy Preserving Network Publication”. In: *Proc. VLDB Endow.* 2.1 (2009), pp. 946–957.

Received July 2022; revised October 2022; accepted November 2022